A Tool for Detection of Co-Occurrences between Design Patterns and Bad Smells

Bruno. L. Sousa[1], **Mariza A. S. Bigonha**[1], **Kecia A. M. Ferreira**[2]

[1]Computer Science Department – Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

[2]Department of Computing – Federal Center for Technological Education of Minas Gerais (CEFET-MG) – Belo Horizonte – MG – Brazil

{bruno.luan.sousa,mariza}@dcc.ufmg.br, kecia@decom.cefetmg.br

Abstract. *A design pattern is a general reusable solution to recurring problems in software design. These solutions are considered good programming practices and aim to produce flexible, extensible and maintainable software. Bad Smells are symptoms present in the software source code that indicate occurrence of possible problems that can impair the quality of the project. Despite that these two structures may have opposite concepts, studies have indicated that they may present relations of co-occurrence during the implementation phase. Since these relationships represent the degenerate structure of a design pattern and consequently reduce the quality of the software, they should be identified and removed. In this paper, we propose Design Pattern Smell, a tool for co-occurrences detection between design patterns and bad smell in software systems. With Design Pattern Smell, the user may identify the artifacts that hold such relationships and the intensity with which they occur in the design patterns analyzed. In addition, Design Pattern Smell may be used as a refactoring guide, since all artifacts detected with co-occurrences are displayed to the user.*
Video: `https://youtu.be/hFwyId9nHnM`

1. Introduction

Design pattern is a general solution applied to a given context in the software design [Gamma et al. 1994], aiming to achieve a high reusability and extensibility in the created modules. These solutions encourage the use of structures composed by inheritance and polymorphism, in order to flex the communication between objects reducing the degree of coupling between modules and improving the software maintainability. The Gang of Four's (GOF) book [Gamma et al. 1994] describes 23 design patterns that are highly used by researchers and developers, and have a major influence in the software engineering.

In contrast, bad smells are symptoms present in a particular region of the software that may indicate a more serious problem in its design or source code [Fowler and Beck 1999]. Fragments of code displaying these symptoms are not considered errors. However, they contribute negatively to the quality of software, increasing its complexity and impairing important features such as modularity, extensibility, reuse and maintainability. Therefore is important to identify these structures and remove them from the source code of a project. In most cases, refactoring techniques are applied to eliminate them.

Although the definitions of design pattern and bad smell are completely opposite, these two structures may have co-occurrence in the software source code. In fact, some researches have investigated this theme and identified these co-occurrence. However, none of these researches have proposed tools to identify them, [Jaafar et al. 2013, Cardoso and Figueiredo 2015, Jaafar et al. 2016, Walter and Alkhaeir 2016]. For instance, analysing the above related works, we perceive that the authors of these studies used their own scripts or manual processing to identify these relationships.

Based on this context, this paper presents a tool, Design Pattern Smell, which supports the detection of co-occurrence between design patterns and bad smell based on computed information from software systems. The tool receives as input XML files containing the design patterns instances and a CSV file containing the code artifacts[1] with bad smell. Design Pattern Smell parses this information and then a data crossing (Vide Section 2.3) to detect the co-occurrences. In addition, it allows the user to apply association rules [Agrawal et al. 1993, Brin et al. 1997] on these data to identify the strength of those relationships. To evaluate the tool, a case study was carried out with five Java software systems from the Qualitas.class Corpus [Terra et al. 2013]. The results of this case study suggest that Design Pattern Smell may be effectively used to support the detection of co-occurrences between design patterns and bad smell in an easy and simple way.

The remainder of this paper is organized as follows. Section 2 describes the main features and technical details of Design Pattern Smell. Section 3 provides the tool's running example. Section 4 presents the evaluation of the tool. Section 5 discusses the related work. Section 6 concludes this paper with suggestion for future work.

2. Design Pattern Smell

This section describes the Design Pattern tool.

2.1. Proposed Approach

Design Pattern Smell is a static analysis tool proposed for identifying co-occurrences of design patterns with bad smell based on information extracted from the source code. The decision to construct it was taken based on the purpose of assisting the identification of existing code that present both design pattern and bad smells, as well as to support an exploratory analysis in order to understand the reasons that contributed to the co-occurrences of these two structures. Although Design Pattern Smell requires information about design patterns and bad smell previously extracted, such information can be easily obtained by tools like: *Desing Pattern Detection* [Tsantalis et al. 2006], *JDeodorant* [Tsantalis et al. 2008], *JSpIRIT* [Vidal et al. 2014] and *RAFTool* [Filó et al. 2014].

Thus, the main objective of Design Pattern Smell is to support the evaluation of software quality by identifying code structures which the presence of a bad smell can degenerate the application of a design pattern. Design Pattern Smell supports the detection of co-occurrences of bad smells at class and method level with 14 design patterns from the GOF catalog [Gamma et al. 1994].

2.2. Main Features

The main features of Design Pattern Smell are described as follows.

[1] Artifact in this paper is a class or method of a system.

Import of Computed Design Pattern Instances. To identify the co-occurrences between design pattern and bad smell, Design Pattern Smell requires that the user imports XML files with design pattern instances in a given system. This input file follows the same format exported by the Design Pattern Detection tool [Tsantalis et al. 2006] and is described on the Design Pattern Smell's website [Sousa et al. 2016].

Import of Computed Artifacts with Bad Smells. Another requirement for identifying co-occurrences is a CSV file containing artifacts with the presence of bad smell. After importing this file, Design Pattern Smell performs the parser of its contents and then crosses the information contained in the CSV file with those in the XML files, identifying the artifacts that have co-occurrence of those two structures. The input file has to be written according to the format specified on the tool's website [Sousa et al. 2016].

Application of Association Rules. The user may apply association rules in the data used, in order to analyze the intensity of co-occurrence between the design patterns and the bad smell. Design Pattern Smell provides a module that applies these association rules automatically, preventing the user from doing this work manually. For the application of the rules, (i) a field is provided where the user enters the number of transactions in the analyzed system and (ii) a panel with 4 types of rules, [Agrawal et al. 1993, Brin et al. 1997], which can be calculated by tool. By default, the 4 rules are pre-selected to be computed. However, the user can filter and calculate only those of his interest. Transaction in association rules, refers to the total number of classes or methods belonging to the system, according to the granularity of the bad smell used.

Result Generation, Visualization and Export. After data crossing, the user may consult the following reports: number of design pattern instances in the system, and amount and information about the artifacts that presented co-occurrences of design patterns with bad smell. In addition, after applying the association rules, the report with the results are displayed to the user. These reports are presented in a data grid view. Design Pattern Smell exports these results to a CSV file for easy analysis and future manipulation.

Data Management. This functionality allows the user to use data from design patterns already stored in Design Pattern Smell to crossing with data from another bad smell. In this case, the user must clear the information regarding bad smell under analysis and perform the import of the other CSV file. This process may also be performed for design patterns when the user wants to change the system under review.

Help. For users unfamiliar with association rules, this functionality describes this method of analysis, as well as the formulas used to calculate relationships, an overview of how to analyze results, and the definition of technical terms used to calculate formulas. In addition, this functionality presents general information about the version of this tool, and provides a link of a video tutorial that presents a running example of the tool.

2.3. Architecture

Design Pattern Smell architecture consists of six internal modules, as shown in Figure 1.

Input Manager. This module is responsible for managing the input files as well as performing the verification and validation of the format required by Design Pattern Smell. In addition, it performs data cleaning and prepares the tool for receiving new information on instances of design patterns or bad smell.

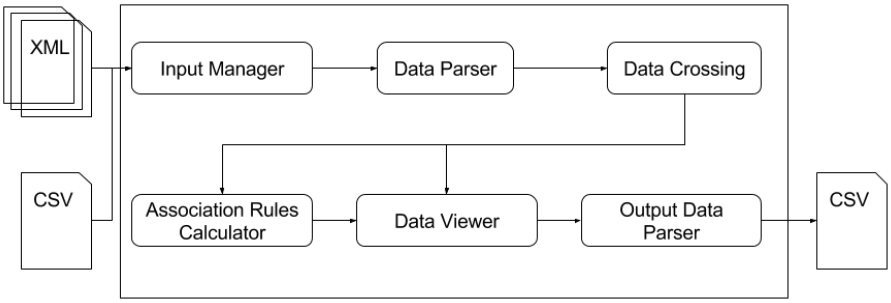

Figure 1. Design Pattern Smell Architecture.

Data Parser. This module is responsible for parsing the information in the input files. It can receive one or more XML files containing computed design pattern instances. Each instance may be composed of several components that play a certain role within these instances. In this module these components are separated and classified according to their granularity: class, method or attribute. The CSV file is parsed, and the information in each row of that file is grouped as a component type and sorted according to the granularity specified by the user. All information extracted from both the XML files and the CSV file is persisted in memory and used in the other modules. The default format for input files is available for viewing on the tool's website [Sousa et al. 2016].

Data Crossing. This module performs the crossing of extracted data in Data Parser to identify artifacts that have co-occurrence of design pattern and bad smell. In this data crossing, it is checked whether each class or method with presence of bad smell is part of some design pattern instance. If yes, the tool lists these artifacts as co-occurrences.

Association Rules Calculator. This module implements the application of association rules in the information provided for the tool, in order to identify the intensity of co-occurrence between the design patterns and bad smell. It uses quantitative information, computed in the Data Crossing module, along with the total amount of system transactions provided by the user to calculate the rules.

Data Viewer. This module allows reporting in a data grid view format. From there, the user can navigate in the list of affected artifacts identified with co-occurrence. In addition, it is possible to issue other types of reports with the information computed by both the Data Crossing module and the Association Rules Calculator, such as: design pattern instances in a system, rate of artifacts affected by co-occurrence, and intensity of co-occurrence identified in each pattern existing in the system.

Output Data Parser. This module performs a parser of the reports issued by the Data Viewer and generates an output CSV file that is stored in a user-defined location on the user machine. This file contains the same information of the data grid view and it may be useful to analyze co-occurrences.

2.4. Implementation

Design Pattern Smell was developed in the Java programming language with JDK 1.7 support and the Java Swing API to create the graphical user interface. Java was chosen because of its portability, and because it is a widespread language both in academia and

industry. To parse the input XML files, we used the JDOM API[2] to interpret and manipulate XML data from Java source code. In order to present the association rules, in the Help option, we used the JLaTeXMath 1.0.3 API[3] to show the mathematical formulas used in each of its metrics. Design Pattern Smell was constructed by means of the Netbeans IDE 8.0.2[4], which provides drag and drop functionality for constructing User interface. Design Pattern Smell is available in version 1.0 on the tool's website [Sousa et al. 2016].

3. Running Example

This section presents an example of using the main features of Design Pattern Smell.

Figure 2 (i) shows the main Design Pattern Smell screen. This screen provides three types of functionality: (i) in the *Import XML Files with Design Pattern Instances*, the user imports the input XML files from a target system. In addition, it is necessary to inform the name of the system, whose imported instances are referent; (ii) in the *Data Crossing*, the user imports a CSV file with classes or methods of the target system that have bad smell. When importing this file, the name of the bad smell and its granularity should be given; (iii) at the top of the main screen there is a menu with four options: *File* allows the user to enter new information regarding other bad smells and design patterns; *Results* provides information and reports on design patterns instances and affected artifacts; *Statistics* allows the user to apply association rules in the data used to identify co-occurrences; and *Help* contains a reference and description of the association rules used by Design Pattern Smell, as well as information about the version of that tool, list of developers and links to the source code and a video tutorial.

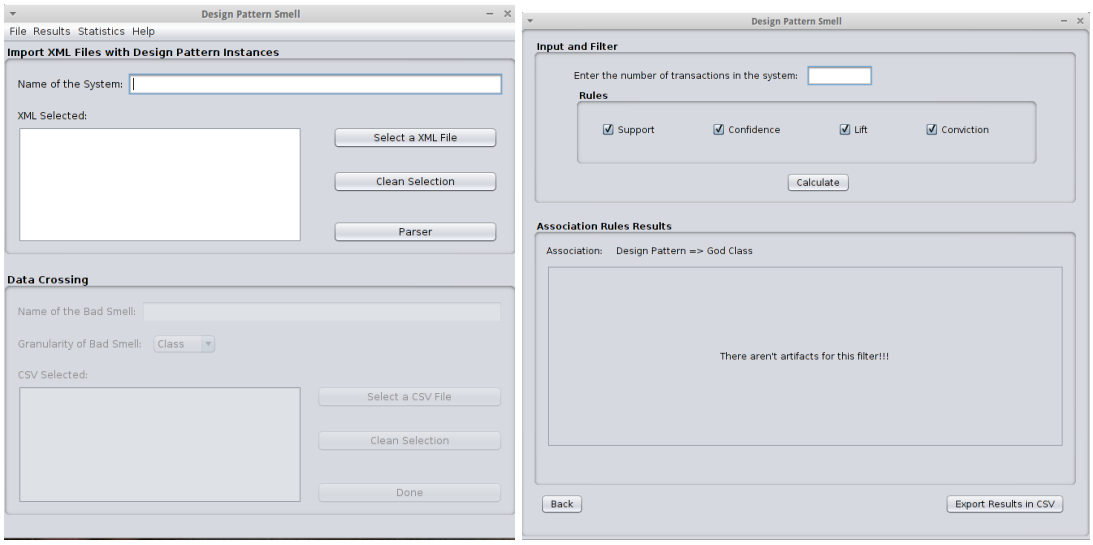

Figure 2. (i) Main Screen to Import XML Files and CSV File; (ii) Screen for Application of Association Rules.

Figure 2 (ii) shows the screen for applying association rules. In this screen, the user enters a value indicating the number of transactions in the target system. Transaction in this context refers to the total number of classes or methods that the target system has.

[2] http://www.jdom.org/
[3] https://forge.scilab.org/index.php/p/jlatexmath/
[4] https://netbeans.org/

The user can still filter the rules that will be calculated. By default, the four rules (support, confidence, lift, and conviction) are pre-selected to be calculated. However, the user can select the rules to be calculated. After calculating the association rules, the values are displayed in *Association Rules Results* in table format that can be exported to a CSV file.

After performing the crossing of the imported data in the screen of Figure 2 (i), Design Pattern Smell allows the user to visualize the number of identified artifacts with co-occurrence of design patterns and bad smell. Figure 3 (i) illustrates the screen responsible for displaying this information. This screen is formed by a table whose lines refer to a particular design pattern. For each line, it is informed the total of artifacts that make up the respective design pattern, the number of artifacts that were affected by bad smell, and the percentage of affected artifacts. These results can be exported to a CSV file via the button in the lower right corner called *Export Results in CSV*.

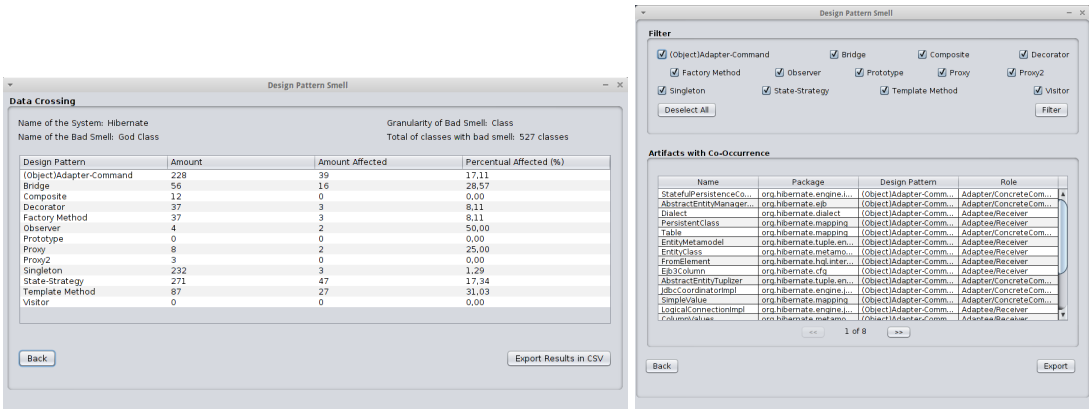

Figure 3. (i)Visualizing the Amount of Affected Artifacts by Co-occurrence; (ii) Screen for Viewing Artifacts Affected by Co-occurrences.

Finally, after crossing the data, the user can visualize the artifacts in which the co-occurrences have been identified. To access this information the user must select the *Artifacts with Co-occurrence* option from the *Results* menu and will be redirected to the screen shown by Figure 3 (ii). This screen displays a table with the list of affected artifacts. Each line refers to an artifact type and displays information such as: name, file (in the case of methods), package, design pattern on which this artifact is referenced, and role exercised within the pattern. In addition, the user may restrict the information displayed on this screen to specific design pattern. To do this, the user have to select the desired options and press the *Filter* button. The results may be exported to a CSV file.

4. Tool's Evaluation

First, Design Pattern Smell was evaluated in a case study to identify co-occurrences between GOF design patterns with bad smells God Class and Long Method [Sousa et al. 2017]. Subsequently, in an empirical study conducted within a research that is underway, we extended this evaluation to the bad smells: Data Class, Feature Envy, and Refused Bequest. In both evaluations, we used a dataset with five Java systems, from the Qualitas.class Corpus [Terra et al. 2013]. From this dataset we extract the GOF design pattern instances and the classes and methods with the presence of bad smell. The results suggest that Design Pattern Smell is able to correctly detect artifacts with co-occurrence

and apply association rules identifying the intensity of the relationships for each design pattern. The results of the evaluation are available on the tool website [Sousa et al. 2016].

5. Related Work

The literature has provided several tools for detection of design patterns and bad smells in software systems. *Design Pattern Detection* [Tsantalis et al. 2006], *PINOT* [Shi and Olsson 2006] and *DPFinder* [Bernardi et al. 2013] are examples of tools that by means of a static analysis in the source code of Java projects, identify GOF design patterns instances. These tools have been used by researchers and developers. *JDeodorant* [Tsantalis et al. 2008], *JSpIRIT* [Vidal et al. 2014] and *RAFTool* [Filó et al. 2014] are examples of tools that detect bad smells. While *JDeodorant* uses an AST approach to symptom identification, *JSpIRIT* and *RAFTool* use a metric-based approach. These tools have also been used by researchers and developers for quality assessment in software projects.

Although there is a wide range of tools that detect design patterns and bad smells in software projects, the literature lacks of tools that identify the co-occurrences of these two structures. Some studies, [Jaafar et al. 2013, Cardoso and Figueiredo 2015, Jaafar et al. 2016, Walter and Alkhaeir 2016], have used their own scripts or manual processing of data extracted from software projects to identify these relationships, without providing any kind of tool to automate this task. To overcome this limitation, we presented the Desisn Pattern Smell that helps the user to detect these relationships in a software project, based on information about design patterns and bad smell computed.

6. Conclusion

This paper presented Design Pattern Smell, a tool for detecting co-occurrences between design patterns and bad smell. This tool receives as input XML files with design pattern instances and a CSV file with code artifacts that have bad smell presence. Design Pattern Smell provides a simple and intuitive interface for selecting input files and provides agility in the detection of artifacts with co-occurrence. In addition, Design Pattern Smell allows the user to apply association rules on the data collected to identify the intensity of the co-occurrences, and to export reports. Design Pattern Smell currently supports co-occurrence detection of bad smells at class and method level with 14 GOF design patterns.

As future work, we want to expand the set of design pattern to all 23 that make up the GOF catalog. In addition, it is desired to add a static analysis module in the source code of Java software systems for automatically identification of design pattern instances, and to add other statistical methods for analysis of intensity of co-occurrences.

7. Acknowledments

This work was sponsored by CAPES.

References

[Agrawal et al. 1993] Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216.

[Bernardi et al. 2013] Bernardi, M. L., Cimitile, M., and Di Lucca, G. A. (2013). A model-driven graph-matching approach for design pattern detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 172–181.

[Brin et al. 1997] Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26:255–264.

[Cardoso and Figueiredo 2015] Cardoso, B. and Figueiredo, E. (2015). Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Proc. of the Conference on Brazilian Symposium on Information Systems*, pages 347–354.

[Filó et al. 2014] Filó, T. G. S., Bigonha, M. A. S., and Ferreira, K. A. M. (2014). Raftool - filtering tool of methods, classes and packages with uncommon measurements of software metrics (in portuguese). In *Proceedings of the X WAMPS 2014*, pages 1–6.

[Fowler and Beck 1999] Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Jaafar et al. 2013] Jaafar, F., Guéhéneuc, Y., Hamel, S., and Khomh, F. (2013). Analysing anti-patterns static relationships with design patterns. *ECEASST*, 59.

[Jaafar et al. 2016] Jaafar, F., Gueheneuc, Y.-G., Hamel, S., Khomh, F., and Zulkernine, M. (2016). Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21(3):896–931.

[Shi and Olsson 2006] Shi, N. and Olsson, R. A. (2006). Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134.

[Sousa et al. 2016] Sousa, B., Bigonha, M., and Ferreira, K. (2016). Design pattern smell. `http://www2.dcc.ufmg.br/laboratorios/llp/Products/indexProducts.html`. Accessed on October 17, 2016.

[Sousa et al. 2017] Sousa, B., Bigonha, M., and Kecia, F. (2017). Evaluating co-occurrence of gof design patterns with god class and long method bad smells. In *Proceedings of the Brazilian Symposium on Information Systems*, pages 1–8. (paper accepted).

[Terra et al. 2013] Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013). Qualitas. class corpus: A compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5):1–4.

[Tsantalis et al. 2008] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *Proc. of the 12th CSMR*, pages 329–331.

[Tsantalis et al. 2006] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909.

[Vidal et al. 2014] Vidal, S. A., Marcos, C., and Díaz-Pace, J. A. (2014). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23:501–532.

[Walter and Alkhaeir 2016] Walter, B. and Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127–142.